LAMPIRAN

1. FaceMask Example

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using OpenCVForUnity.CoreModule;

namespace FaceMaskExample
{
    /// <summary>
    /// FaceMask Example
    /// </summary>
    public class FaceMaskExample : MonoBehaviour
    {
        public Text exampleTitle;
        public Text versionInfo;
        public ScrollRect scrollRect;
        static float verticalNormalizedPosition = 1f;

        // Use this for initialization
        void Start ()
        {
            exampleTitle.text = "FaceMask Example " + Application.version;

            versionInfo.text = Core.NATIVE_LIBRARY_NAME + " " +
OpenCVForUnity.UnityUtils.Utils.getVersion () + " (" + Core.VERSION + ")";
            versionInfo.text += " / " + "dlibfacelandmarkdetector" + " " +
DlibFaceLandmarkDetector.UnityUtils.Utils.getVersion ();
            versionInfo.text += " / UnityEditor " + Application.unityVersion;
            versionInfo.text += " / ";

            #if UNITY_EDITOR
            versionInfo.text += "Editor";
            #elif UNITY_STANDALONE_WIN
            versionInfo.text += "Windows";
            #elif UNITY_STANDALONE_OSX
            versionInfo.text += "Mac OSX";
            #elif UNITY_STANDALONE_LINUX
            versionInfo.text += "Linux";
            #elif UNITY_ANDROID
            versionInfo.text += "Android";
```

```
    #elif UNITY_IOS
    versionInfo.text += "iOS";
    #elif UNITY_WSA
    versionInfo.text += "WSA";
    #elif UNITY_WEBGL
    versionInfo.text += "WebGL";
    #endif
    versionInfo.text += " ";
    #if ENABLE_MONO
    versionInfo.text += "Mono";
    #elif ENABLE_IL2CPP
    versionInfo.text += "IL2CPP";
    #elif ENABLE_DOTNET
    versionInfo.text += ".NET";
    #endif

    scrollRect.verticalNormalizedPosition = verticalNormalizedPosition;
}

// Update is called once per frame
void Update ()
{

}

public void OnScrollRectValueChanged ()
{
    verticalNormalizedPosition = scrollRect.verticalNormalizedPosition;
}


public void OnShowLicenseButtonClick ()
{
    SceneManager.LoadScene ("ShowLicense");
}

public void OnTexture2DFaceMaskExampleButtonClick ()
{
    SceneManager.LoadScene ("Texture2DFaceMaskExample");
}

public void OnVideoCaptureFaceMaskExampleButtonClick ()
{
    SceneManager.LoadScene ("VideoCaptureFaceMaskExample");
}
```

```
        public void OnWebCamTextureFaceMaskExampleButtonClick ()
        {
            SceneManager.LoadScene ("WebCamTextureFaceMaskExample");
        }

        public void OnWebCamTextureFaceMaskAdditionalExampleButtonClick ()
        {
            SceneManager.LoadScene
("WebCamTextureFaceMaskAdditionalExample");
        }
    }
}

using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;

namespace FaceMaskExample
{
    /// <summary>
    /// Show License
    /// </summary>
    public class ShowLicense : MonoBehaviour
    {
        // Use this for initialization
        void Start ()
        {

        }

        // Update is called once per frame
        void Update ()
        {

        }

        public void OnBackButtonClick ()
        {
            SceneManager.LoadScene ("FaceMaskExample");
        }
    }
}
```

2. Font Menu

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FrontMenu : MonoBehaviour
{
   public void GotoWardah(string scenename)
   {
      Application.LoadLevel(scenename);
   }

   public void doExitGame()
   {
      Application.Quit();
   }
}

using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using DlibFaceLandmarkDetector;
using OpenCVForUnity.RectangleTrack;
using OpenCVForUnity.UnityUtils.Helper;
using OpenCVForUnity.CoreModule;
using OpenCVForUnity.ObjdetectModule;
using OpenCVForUnity.ImgprocModule;
using Rect = OpenCVForUnity.CoreModule.Rect;

namespace FaceMaskExample
{
   /// <summary>
   /// WebCamTexture FaceMask Additional Example
   /// </summary>
   [RequireComponent(typeof(WebCamTextureToMatHelper),
typeof(TrackedMeshOverlay))]
   public class WebCamTextureFaceMaskAdditionalExample : MonoBehaviour
   {
      [HeaderAttribute("Additional FaceMask Option")]
```

```
/// <summary>
/// Determines if make both eyes transparent.
/// </summary>
public bool makeBothEyesTransparent = true;

/// <summary>
/// Determines if make the mouth transparent.
/// </summary>
public bool makeMouthTransparent = true;

/// <summary>
/// Determines if extend the forehead.
/// </summary>
public bool extendForehead = true;

[Space(15)]

[HeaderAttribute("FaceMaskData")]

/// <summary>
/// The face mask data list.
/// </summary>
public List<FaceMaskData> faceMaskDatas;

[HeaderAttribute("Option")]

/// <summary>
/// Determines if use dlib face detector.
/// </summary>
public bool useDlibFaceDetecter = false;

/// <summary>
/// The use dlib face detecter toggle.
/// </summary>
public Toggle useDlibFaceDetecterToggle;

/// <summary>
/// Determines if enables noise filter.
/// </summary>
public bool enableNoiseFilter = true;

/// <summary>
/// The enable noise filter toggle.
/// </summary>
public Toggle enableNoiseFilterToggle;
```

```
/// <summary>
/// Determines if enables color correction.
/// </summary>
public bool enableColorCorrection = true;

/// <summary>
/// The enable color correction toggle.
/// </summary>
public Toggle enableColorCorrectionToggle;

/// <summary>
/// Determines if filters non frontal faces.
/// </summary>
public bool filterNonFrontalFaces = false;

/// <summary>
/// The filter non frontal faces toggle.
/// </summary>
public Toggle filterNonFrontalFacesToggle;

/// <summary>
/// The frontal face rate lower limit.
/// </summary>
[Range(0.0f, 1.0f)]
public float frontalFaceRateLowerLimit = 0.85f;

/// <summary>
/// Determines if displays face rects.
/// </summary>
public bool displayFaceRects = false;

/// <summary>
/// The toggle for switching face rects display state
/// </summary>
public Toggle displayFaceRectsToggle;

/// <summary>
/// Determines if displays debug face points.
/// </summary>
public bool displayDebugFacePoints = false;

/// <summary>
/// The toggle for switching debug face points display state.
/// </summary>
public Toggle displayDebugFacePointsToggle;
```

```
/// <summary>
/// The gray mat.
/// </summary>
Mat grayMat;

/// <summary>
/// The texture.
/// </summary>
Texture2D texture;

/// <summary>
/// The cascade.
/// </summary>
CascadeClassifier cascade;

/// <summary>
/// The detection based tracker.
/// </summary>
RectangleTracker rectangleTracker;

/// <summary>
/// The web cam texture to mat helper.
/// </summary>
WebCamTextureToMatHelper webCamTextureToMatHelper;

/// <summary>
/// The face landmark detector.
/// </summary>
FaceLandmarkDetector faceLandmarkDetector;

/// <summary>
/// The mean points filter dictionary.
/// </summary>
Dictionary<int, LowPassPointsFilter> lowPassFilterDict;

/// <summary>
/// The optical flow points filter dictionary.
/// </summary>
Dictionary<int, OFPointsFilter> opticalFlowFilterDict;

/// <summary>
/// The face mask color corrector.
/// </summary>
FaceMaskColorCorrector faceMaskColorCorrector;

/// <summary>
```

```
/// The frontal face checker.
/// </summary>
FrontalFaceChecker frontalFaceChecker;

/// <summary>
/// The mesh overlay.
/// </summary>
TrackedMeshOverlay meshOverlay;

/// <summary>
/// The Shader.PropertyToID for "_Fade".
/// </summary>
int shader_FadeID;

/// <summary>
/// The Shader.PropertyToID for "_ColorCorrection".
/// </summary>
int shader_ColorCorrectionID;

/// <summary>
/// The Shader.PropertyToID for "_LUTTex".
/// </summary>
int shader_LUTTexID;

/// <summary>
/// The face mask texture.
/// </summary>
Texture2D faceMaskTexture;

/// <summary>
/// The face mask mat.
/// </summary>
Mat faceMaskMat;

/// <summary>
/// The index number of face mask data.
/// </summary>
int faceMaskDataIndex = 0;

/// <summary>
/// The detected face rect in mask mat.
/// </summary>
UnityEngine.Rect faceRectInMask;

/// <summary>
/// The detected face landmark points in mask mat.
```

```
        /// </summary>
        List<Vector2> faceLandmarkPointsInMask;

        /// <summary>
        /// The haarcascade_frontalface_alt_xml_filepath.
        /// </summary>
        string haarcascade_frontalface_alt_xml_filepath;

        /// <summary>
        /// The sp_human_face_68_dat_filepath.
        /// </summary>
        string sp_human_face_68_for_mobile_dat_filepath;

        /// <summary>
        /// The FPS monitor.
        /// </summary>
        FpsMonitor fpsMonitor;




#if UNITY_WEBGL && !UNITY_EDITOR
        IEnumerator getFilePath_Coroutine;
#endif

        // Use this for initialization
        void Start()
        {
            fpsMonitor = GetComponent<FpsMonitor>();

            webCamTextureToMatHelper =
gameObject.GetComponent<WebCamTextureToMatHelper>();

#if UNITY_WEBGL && !UNITY_EDITOR
            getFilePath_Coroutine = GetFilePath ();
            StartCoroutine (getFilePath_Coroutine);
#else
            haarcascade_frontalface_alt_xml_filepath =
OpenCVForUnity.UnityUtils.Utils.getFilePath("haarcascade_frontalface_alt.xml"
);
            sp_human_face_68_for_mobile_dat_filepath =
DlibFaceLandmarkDetector.UnityUtils.Utils.getFilePath("sp_human_face_68_for
_mobile.dat");
```

```
            Run();
#endif
        }

#if UNITY_WEBGL && !UNITY_EDITOR
        private IEnumerator GetFilePath ()
        {
            var getFilePathAsync_0_Coroutine =
OpenCVForUnity.UnityUtils.Utils.getFilePathAsync
("haarcascade_frontalface_alt.xml", (result) => {
                haarcascade_frontalface_alt_xml_filepath = result;
            });
            yield return getFilePathAsync_0_Coroutine;

            var getFilePathAsync_1_Coroutine =
DlibFaceLandmarkDetector.UnityUtils.Utils.getFilePathAsync
("sp_human_face_68_for_mobile.dat", (result) => {
                sp_human_face_68_for_mobile_dat_filepath = result;
            });
            yield return getFilePathAsync_1_Coroutine;

            getFilePath_Coroutine = null;

            Run ();
        }
#endif

        private void Run()
        {
            meshOverlay = this.GetComponent<TrackedMeshOverlay>();

            // Customize face mask.
            GameObject newBaseObject = Instantiate(meshOverlay.baseObject);
            newBaseObject.name = "CustomFaceMaskTrackedMesh";
            TrackedMesh tm = newBaseObject.GetComponent<TrackedMesh>();

            if (extendForehead)
            {
                ExtendForehead(tm.meshFilter.mesh);
            }

            Texture alphaMask = tm.material.GetTexture("_MaskTex");
            Vector2[] uv = tm.meshFilter.sharedMesh.uv2;
            Texture2D newAlphaMask =
CreateFaceMaskAlphaMaskTexture(alphaMask.width, alphaMask.height, uv,
makeBothEyesTransparent, makeMouthTransparent);
```

```
            tm.material.SetTexture("_MaskTex", newAlphaMask);

            meshOverlay.baseObject = newBaseObject;


            shader_FadeID = Shader.PropertyToID("_Fade");
            shader_ColorCorrectionID = Shader.PropertyToID("_ColorCorrection");
            shader_LUTTexID = Shader.PropertyToID("_LUTTex");

            rectangleTracker = new RectangleTracker();

            faceLandmarkDetector = new
FaceLandmarkDetector(sp_human_face_68_for_mobile_dat_filepath);

            lowPassFilterDict = new Dictionary<int, LowPassPointsFilter>();
            opticalFlowFilterDict = new Dictionary<int, OFPointsFilter>();

            faceMaskColorCorrector = new FaceMaskColorCorrector();

#if UNITY_ANDROID && !UNITY_EDITOR
            // Avoids the front camera low light issue that occurs in only some
Android devices (e.g. Google Pixel, Pixel2).
            webCamTextureToMatHelper.avoidAndroidFrontCameraLowLightIssue =
true;
#endif
            webCamTextureToMatHelper.Initialize();

            displayFaceRectsToggle.isOn = displayFaceRects;
            useDlibFaceDetecterToggle.isOn = useDlibFaceDetecter;
            enableNoiseFilterToggle.isOn = enableNoiseFilter;
            enableColorCorrectionToggle.isOn = enableColorCorrection;
            filterNonFrontalFacesToggle.isOn = filterNonFrontalFaces;
            displayDebugFacePointsToggle.isOn = displayDebugFacePoints;
        }

        /// <summary>
        /// Raises the web cam texture to mat helper initialized event.
        /// </summary>
        public void OnWebCamTextureToMatHelperInitialized()
        {
            Debug.Log("OnWebCamTextureToMatHelperInitialized");

            Mat webCamTextureMat = webCamTextureToMatHelper.GetMat();

            texture = new Texture2D(webCamTextureMat.cols(),
webCamTextureMat.rows(), TextureFormat.RGBA32, false);
```

```
        gameObject.transform.localScale = new
Vector3(webCamTextureMat.cols(), webCamTextureMat.rows(), 1);
        Debug.Log("Screen.width " + Screen.width + " Screen.height " +
Screen.height + " Screen.orientation " + Screen.orientation);

        if (fpsMonitor != null)
        {
           fpsMonitor.Add("width", webCamTextureMat.width().ToString());
           fpsMonitor.Add("height", webCamTextureMat.height().ToString());
           fpsMonitor.Add("orientation", Screen.orientation.ToString());
        }


        float width = gameObject.transform.localScale.x;
        float height = gameObject.transform.localScale.y;

        float widthScale = (float)Screen.width / width;
        float heightScale = (float)Screen.height / height;
        if (widthScale < heightScale)
        {
           Camera.main.orthographicSize = (width * (float)Screen.height /
(float)Screen.width) / 2;
        }
        else
        {
           Camera.main.orthographicSize = height / 2;
        }

        gameObject.GetComponent<Renderer>().material.mainTexture = texture;

        grayMat = new Mat(webCamTextureMat.rows(),
webCamTextureMat.cols(), CvType.CV_8UC1);
        cascade = new
CascadeClassifier(haarcascade_frontalface_alt_xml_filepath);
        //          if (cascade.empty ()) {
        //              Debug.LogError ("cascade file is not loaded.Please copy from
“FaceTrackerExample/StreamingAssets/” to “Assets/StreamingAssets/” folder. ");
        //          }

        frontalFaceChecker = new FrontalFaceChecker(width, height);

        meshOverlay.UpdateOverlayTransform(gameObject.transform);

        OnChangeFaceMaskButtonClick();
```

```
    }

    /// <summary>
    /// Raises the web cam texture to mat helper disposed event.
    /// </summary>
    public void OnWebCamTextureToMatHelperDisposed()
    {
        Debug.Log("OnWebCamTextureToMatHelperDisposed");

        grayMat.Dispose();

        if (texture != null)
        {
            Texture2D.Destroy(texture);
            texture = null;
        }

        rectangleTracker.Reset();
        meshOverlay.Reset();

        foreach (var key in lowPassFilterDict.Keys)
        {
            lowPassFilterDict[key].Dispose();
        }
        lowPassFilterDict.Clear();
        foreach (var key in opticalFlowFilterDict.Keys)
        {
            opticalFlowFilterDict[key].Dispose();
        }
        opticalFlowFilterDict.Clear();

        faceMaskColorCorrector.Reset();

        frontalFaceChecker.Dispose();
    }

    /// <summary>
    /// Raises the web cam texture to mat helper error occurred event.
    /// </summary>
    /// <param name="errorCode">Error code.</param>
    public void
OnWebCamTextureToMatHelperErrorOccurred(WebCamTextureToMatHelper.E
rrorCode errorCode)
    {
        Debug.Log("OnWebCamTextureToMatHelperErrorOccurred " +
errorCode);
```

```
    }

    // Update is called once per frame
    void Update()
    {
        if (webCamTextureToMatHelper.IsPlaying() &&
webCamTextureToMatHelper.DidUpdateThisFrame())
        {

            Mat rgbaMat = webCamTextureToMatHelper.GetMat();

            // detect faces.
            List<Rect> detectResult = new List<Rect>();
            if (useDlibFaceDetecter)
            {
                OpenCVForUnityUtils.SetImage(faceLandmarkDetector, rgbaMat);
                List<UnityEngine.Rect> result = faceLandmarkDetector.Detect();

                foreach (var unityRect in result)
                {
                    detectResult.Add(new Rect((int)unityRect.x, (int)unityRect.y,
(int)unityRect.width, (int)unityRect.height));
                }
            }
            else
            {
                // convert image to greyscale.
                Imgproc.cvtColor(rgbaMat, grayMat,
Imgproc.COLOR_RGBA2GRAY);

                using (Mat equalizeHistMat = new Mat())
                using (MatOfRect faces = new MatOfRect())
                {
                    Imgproc.equalizeHist(grayMat, equalizeHistMat);

                    cascade.detectMultiScale(equalizeHistMat, faces, 1.1f, 2, 0 |
Objdetect.CASCADE_SCALE_IMAGE, new Size(equalizeHistMat.cols() * 0.15,
equalizeHistMat.cols() * 0.15), new Size());

                    detectResult = faces.toList();
                }

                // corrects the deviation of a detection result between OpenCV and
Dlib.
                foreach (Rect r in detectResult)
                {
```

```
                r.y += (int)(r.height * 0.1f);
            }
        }

        // face tracking.
        rectangleTracker.UpdateTrackedObjects(detectResult);
        List<TrackedRect> trackedRects = new List<TrackedRect>();
        rectangleTracker.GetObjects(trackedRects, true);

        // create noise filter.
        foreach (var openCVRect in trackedRects)
        {
            if (openCVRect.state == TrackedState.NEW)
            {
                if (!lowPassFilterDict.ContainsKey(openCVRect.id))
                    lowPassFilterDict.Add(openCVRect.id, new
LowPassPointsFilter((int)faceLandmarkDetector.GetShapePredictorNumParts()));
                if (!opticalFlowFilterDict.ContainsKey(openCVRect.id))
                    opticalFlowFilterDict.Add(openCVRect.id, new
OFPointsFilter((int)faceLandmarkDetector.GetShapePredictorNumParts()));
            }
            else if (openCVRect.state == TrackedState.DELETED)
            {
                if (lowPassFilterDict.ContainsKey(openCVRect.id))
                {
                    lowPassFilterDict[openCVRect.id].Dispose();
                    lowPassFilterDict.Remove(openCVRect.id);
                }
                if (opticalFlowFilterDict.ContainsKey(openCVRect.id))
                {
                    opticalFlowFilterDict[openCVRect.id].Dispose();
                    opticalFlowFilterDict.Remove(openCVRect.id);
                }
            }
        }

        // create LUT texture.
        foreach (var openCVRect in trackedRects)
        {
            if (openCVRect.state == TrackedState.NEW)
            {
                faceMaskColorCorrector.CreateLUTTex(openCVRect.id);
            }
            else if (openCVRect.state == TrackedState.DELETED)
            {
                faceMaskColorCorrector.DeleteLUTTex(openCVRect.id);
```

```
            }
        }

        // detect face landmark points.
        OpenCVForUnityUtils.SetImage(faceLandmarkDetector, rgbaMat);
        List<List<Vector2>> landmarkPoints = new List<List<Vector2>>();
        for (int i = 0; i < trackedRects.Count; i++)
        {
            TrackedRect tr = trackedRects[i];
            UnityEngine.Rect rect = new UnityEngine.Rect(tr.x, tr.y, tr.width,
tr.height);

            List<Vector2> points =
faceLandmarkDetector.DetectLandmark(rect);

            // apply noise filter.
            if (enableNoiseFilter)
            {
                if (tr.state > TrackedState.NEW && tr.state <
TrackedState.DELETED)
                {
                    opticalFlowFilterDict[tr.id].Process(rgbaMat, points, points);
                    lowPassFilterDict[tr.id].Process(rgbaMat, points, points);
                }
            }

            if (extendForehead)
            {
                AddForeheadPoints(points);
            }

            landmarkPoints.Add(points);
        }

        // face masking.
        if (faceMaskTexture != null && landmarkPoints.Count >= 1)
        { // Apply face masking between detected faces and a face mask image.

            float maskImageWidth = faceMaskTexture.width;
            float maskImageHeight = faceMaskTexture.height;

            TrackedRect tr;

            for (int i = 0; i < trackedRects.Count; i++)
            {
                tr = trackedRects[i];
```

```
                if (tr.state == TrackedState.NEW)
                {
                    meshOverlay.CreateObject(tr.id, faceMaskTexture);
                }
                if (tr.state < TrackedState.DELETED)
                {
                    MaskFace(meshOverlay, tr, landmarkPoints[i],
faceLandmarkPointsInMask, maskImageWidth, maskImageHeight);

                    if (enableColorCorrection)
                    {
                        CorrectFaceMaskColor(tr.id, faceMaskMat, rgbaMat,
faceLandmarkPointsInMask, landmarkPoints[i]);
                    }
                }
                else if (tr.state == TrackedState.DELETED)
                {
                    meshOverlay.DeleteObject(tr.id);
                }
            }
        }
        else if (landmarkPoints.Count >= 1)
        { // Apply face masking between detected faces.

            float maskImageWidth = texture.width;
            float maskImageHeight = texture.height;

            TrackedRect tr;

            for (int i = 0; i < trackedRects.Count; i++)
            {
                tr = trackedRects[i];

                if (tr.state == TrackedState.NEW)
                {
                    meshOverlay.CreateObject(tr.id, texture);
                }
                if (tr.state < TrackedState.DELETED)
                {
                  MaskFace(meshOverlay, tr, landmarkPoints[i],
landmarkPoints[0],    maskImageWidth, maskImageHeight);

                    if (enableColorCorrection)
                    {
```

```
                        CorrectFaceMaskColor(tr.id, rgbaMat, rgbaMat,
landmarkPoints[0], landmarkPoints[i]);
                    }
                }
                else if (tr.state == TrackedState.DELETED)
                {
                    meshOverlay.DeleteObject(tr.id);
                }
            }
        }

        // draw face rects.
        if (displayFaceRects)
        {
            for (int i = 0; i < detectResult.Count; i++)
            {
                UnityEngine.Rect rect = new UnityEngine.Rect(detectResult[i].x,
detectResult[i].y, detectResult[i].width, detectResult[i].height);
                OpenCVForUnityUtils.DrawFaceRect(rgbaMat, rect, new
Scalar(255, 0, 0, 255), 2);
            }

            for (int i = 0; i < trackedRects.Count; i++)
            {
                UnityEngine.Rect rect = new UnityEngine.Rect(trackedRects[i].x,
trackedRects[i].y, trackedRects[i].width, trackedRects[i].height);
                OpenCVForUnityUtils.DrawFaceRect(rgbaMat, rect, new
Scalar(255, 255, 0, 255), 2);
                //                  Imgproc.putText (rgbaMat, " " +
frontalFaceChecker.GetFrontalFaceAngles (landmarkPoints [i]), new Point
(rect.xMin, rect.yMin - 10), Imgproc.FONT_HERSHEY_SIMPLEX, 0.5, new
Scalar (255, 255, 255, 255), 2, Imgproc.LINE_AA, false);
                //                  Imgproc.putText (rgbaMat, " " +
frontalFaceChecker.GetFrontalFaceRate (landmarkPoints [i]), new Point
(rect.xMin, rect.yMin - 10), Imgproc.FONT_HERSHEY_SIMPLEX, 0.5, new
Scalar (255, 255, 255, 255), 2, Imgproc.LINE_AA, false);
            }
        }

        // draw face points.
        if (displayDebugFacePoints)
        {
            for (int i = 0; i < landmarkPoints.Count; i++)
            {
                DrawFaceLandmark(rgbaMat, landmarkPoints[i], new Scalar(0,
255, 0, 255), 2);
```

```
            }
        }


        // display face mask image.
        if (faceMaskTexture != null && faceMaskMat != null)
        {

            if (displayFaceRects)
            {
                OpenCVForUnityUtils.DrawFaceRect(faceMaskMat,
faceRectInMask, new Scalar(255, 0, 0, 255), 2);
            }
            if (displayDebugFacePoints)
            {
                DrawFaceLandmark(faceMaskMat, faceLandmarkPointsInMask,
new Scalar(0, 255, 0, 255), 2);
            }

            float scale = (rgbaMat.width() / 4f) / faceMaskMat.width();
            float tx = rgbaMat.width() - faceMaskMat.width() * scale;
            float ty = 0.0f;
            Mat trans = new Mat(2, 3, CvType.CV_32F);//1.0, 0.0, tx, 0.0, 1.0,
ty);
            trans.put(0, 0, scale);
            trans.put(0, 1, 0.0f);
            trans.put(0, 2, tx);
            trans.put(1, 0, 0.0f);
            trans.put(1, 1, scale);
            trans.put(1, 2, ty);

            Imgproc.warpAffine(faceMaskMat, rgbaMat, trans, rgbaMat.size(),
Imgproc.INTER_LINEAR, Core.BORDER_TRANSPARENT, new Scalar(0));

            if (displayFaceRects || displayDebugFacePointsToggle)

OpenCVForUnity.UnityUtils.Utils.texture2DToMat(faceMaskTexture,
faceMaskMat);
        }

        //              Imgproc.putText (rgbaMat, "W:" + rgbaMat.width () + " H:"
+ rgbaMat.height () + " SO:" + Screen.orientation, new Point (5, rgbaMat.rows ()
- 10), Imgproc.FONT_HERSHEY_SIMPLEX, 0.5, new Scalar (255, 255, 255,
255), 1, Imgproc.LINE_AA, false);
```

```
            OpenCVForUnity.UnityUtils.Utils.fastMatToTexture2D(rgbaMat,
texture);
        }
    }

    private void MaskFace(TrackedMeshOverlay meshOverlay, TrackedRect tr,
List<Vector2> landmarkPoints, List<Vector2> landmarkPointsInMaskImage,
float maskImageWidth = 0, float maskImageHeight = 0)
    {
        float imageWidth = meshOverlay.width;
        float imageHeight = meshOverlay.height;

        if (maskImageWidth == 0)
            maskImageWidth = imageWidth;

        if (maskImageHeight == 0)
            maskImageHeight = imageHeight;

        TrackedMesh tm = meshOverlay.GetObjectById(tr.id);

        Vector3[] vertices = tm.meshFilter.mesh.vertices;
        if (vertices.Length == landmarkPoints.Count)
        {
            for (int j = 0; j < vertices.Length; j++)
            {
                vertices[j].x = landmarkPoints[j].x / imageWidth - 0.5f;
                vertices[j].y = 0.5f - landmarkPoints[j].y / imageHeight;
            }
        }
        Vector2[] uv = tm.meshFilter.mesh.uv;
        if (uv.Length == landmarkPointsInMaskImage.Count)
        {
            for (int jj = 0; jj < uv.Length; jj++)
            {
                uv[jj].x = landmarkPointsInMaskImage[jj].x / maskImageWidth;
                uv[jj].y = (maskImageHeight - landmarkPointsInMaskImage[jj].y) /
maskImageHeight;
            }
        }
        meshOverlay.UpdateObject(tr.id, vertices, null, uv);

        if (tr.numFramesNotDetected > 3)
        {
            tm.sharedMaterial.SetFloat(shader_FadeID, 1f);
        }
```

```
            else if (tr.numFramesNotDetected > 0 && tr.numFramesNotDetected <=
3)
            {
                tm.sharedMaterial.SetFloat(shader_FadeID, 0.3f + (0.7f / 4f) *
tr.numFramesNotDetected);
            }
            else
            {
                tm.sharedMaterial.SetFloat(shader_FadeID, 0.3f);
            }

            if (enableColorCorrection)
            {
                tm.sharedMaterial.SetFloat(shader_ColorCorrectionID, 1f);
            }
            else
            {
                tm.sharedMaterial.SetFloat(shader_ColorCorrectionID, 0f);
            }

            // filter non frontal faces.
            if (filterNonFrontalFaces &&
frontalFaceChecker.GetFrontalFaceRate(landmarkPoints) <
frontalFaceRateLowerLimit)
            {
                tm.sharedMaterial.SetFloat(shader_FadeID, 1f);
            }
        }

        private void CorrectFaceMaskColor(int id, Mat src, Mat dst, List<Vector2>
src_landmarkPoints, List<Vector2> dst_landmarkPoints)
        {
            Texture2D LUTTex = faceMaskColorCorrector.UpdateLUTTex(id, src,
dst, src_landmarkPoints, dst_landmarkPoints);
            TrackedMesh tm = meshOverlay.GetObjectById(id);
            tm.sharedMaterial.SetTexture(shader_LUTTexID, LUTTex);
        }

        /// <summary>
        /// Raises the destroy event.
        /// </summary>
        void OnDestroy()
        {
            webCamTextureToMatHelper.Dispose();

            if (cascade != null)
```

```
            cascade.Dispose();

        if (rectangleTracker != null)
            rectangleTracker.Dispose();

        if (faceLandmarkDetector != null)
            faceLandmarkDetector.Dispose();

        foreach (var key in lowPassFilterDict.Keys)
        {
            lowPassFilterDict[key].Dispose();
        }
        lowPassFilterDict.Clear();
        foreach (var key in opticalFlowFilterDict.Keys)
        {
            opticalFlowFilterDict[key].Dispose();
        }
        opticalFlowFilterDict.Clear();

        if (faceMaskColorCorrector != null)
            faceMaskColorCorrector.Dispose();

#if UNITY_WEBGL && !UNITY_EDITOR
        if (getFilePath_Coroutine != null) {
            StopCoroutine (getFilePath_Coroutine);
            ((IDisposable)getFilePath_Coroutine).Dispose ();
        }
#endif
    }

    /// <summary>
    /// Raises the back button click event.
    /// </summary>
    public void OnBackButtonClick()
    {
        SceneManager.LoadScene("FrontMenu");
    }

    /// <summary>
    /// Raises the play button click event.
    /// </summary>
    public void OnPlayButtonClick()
    {
        webCamTextureToMatHelper.Play();
    }
```

```
        /// <summary>
        /// Raises the pause button click event.
        /// </summary>
        public void OnPauseButtonClick()
        {
            webCamTextureToMatHelper.Pause();
        }

        /// <summary>
        /// Raises the change camera button click event.
        /// </summary>
        public void OnChangeCameraButtonClick()
        {
            webCamTextureToMatHelper.requestedIsFrontFacing =
!webCamTextureToMatHelper.IsFrontFacing();
        }

        /// <summary>
        /// Raises the use Dlib face detector toggle value changed event.
        /// </summary>
        public void OnUseDlibFaceDetecterToggleValueChanged()
        {
            if (useDlibFaceDetecterToggle.isOn)
            {
                useDlibFaceDetecter = true;
            }
            else
            {
                useDlibFaceDetecter = false;
            }
        }

        /// <summary>
        /// Raises the enable noise filter toggle value changed event.
        /// </summary>
        public void OnEnableNoiseFilterToggleValueChanged()
        {
            if (enableNoiseFilterToggle.isOn)
            {
                enableNoiseFilter = true;
                foreach (var key in lowPassFilterDict.Keys)
                {
                    lowPassFilterDict[key].Reset();
                }
                foreach (var key in opticalFlowFilterDict.Keys)
                {
```

```
            opticalFlowFilterDict[key].Reset();
        }
    }
    else
    {
        enableNoiseFilter = false;
    }
}

/// <summary>
/// Raises the enable color correction toggle value changed event.
/// </summary>
public void OnEnableColorCorrectionToggleValueChanged()
{
    if (enableColorCorrectionToggle.isOn)
    {
        enableColorCorrection = true;
    }
    else
    {
        enableColorCorrection = false;
    }
}

/// <summary>
/// Raises the filter non frontal faces toggle value changed event.
/// </summary>
public void OnFilterNonFrontalFacesToggleValueChanged()
{
    if (filterNonFrontalFacesToggle.isOn)
    {
        filterNonFrontalFaces = true;
    }
    else
    {
        filterNonFrontalFaces = false;
    }
}

/// <summary>
/// Raises the display face rects toggle value changed event.
/// </summary>
public void OnDisplayFaceRectsToggleValueChanged()
{
    if (displayFaceRectsToggle.isOn)
    {
```

```
            displayFaceRects = true;
        }
        else
        {
            displayFaceRects = false;
        }
    }

    /// <summary>
    /// Raises the display debug face points toggle value changed event.
    /// </summary>
    public void OnDisplayDebugFacePointsToggleValueChanged()
    {
        if (displayDebugFacePointsToggle.isOn)
        {
            displayDebugFacePoints = true;
        }
        else
        {
            displayDebugFacePoints = false;
        }
    }

    /// <summary>
    /// Raises the change face mask button click event.
    /// </summary>
    public void OnChangeFaceMaskButtonClick()
    {
        RemoveFaceMask();

        if (faceMaskDatas.Count == 0)
            return;

        FaceMaskData maskData = faceMaskDatas[faceMaskDataIndex];
        faceMaskDataIndex = (faceMaskDataIndex < faceMaskDatas.Count - 1) ?
faceMaskDataIndex + 1 : 0;

        if (maskData == null)
        {
            Debug.LogError("maskData == null");
            return;
        }

        if (maskData.image == null)
        {
            Debug.LogError("image == null");
```

```
            return;
        }

        if (maskData.landmarkPoints.Count != 68)
        {
            Debug.LogError("landmarkPoints.Count != 68");
            return;
        }

        faceMaskTexture = maskData.image;
        faceMaskMat = new Mat(faceMaskTexture.height,
faceMaskTexture.width, CvType.CV_8UC4);
        OpenCVForUnity.UnityUtils.Utils.texture2DToMat(faceMaskTexture,
faceMaskMat);

        if (maskData.isDynamicMode)
        {
            faceRectInMask = DetectFace(faceMaskMat);
            faceLandmarkPointsInMask =
DetectFaceLandmarkPoints(faceMaskMat, faceRectInMask);

            maskData.faceRect = faceRectInMask;
            maskData.landmarkPoints = faceLandmarkPointsInMask;
        }
        else
        {
            faceRectInMask = maskData.faceRect;
            faceLandmarkPointsInMask = maskData.landmarkPoints;
        }

        if (extendForehead)
        {
            List<Vector2> newLandmarkPointsInMask = new
List<Vector2>(faceLandmarkPointsInMask);
            AddForeheadPoints(newLandmarkPointsInMask);
            faceLandmarkPointsInMask = newLandmarkPointsInMask;
        }

        if (faceRectInMask.width == 0 && faceRectInMask.height == 0)
        {
            RemoveFaceMask();
            Debug.LogError("A face could not be detected from the input image.");
        }

        enableColorCorrectionToggle.isOn = maskData.enableColorCorrection;
    }
```

```
    /// <summary>
    /// Raises the scan face mask button click event.
    /// </summary>
    public void OnScanFaceMaskButtonClick()
    {
        RemoveFaceMask();

        // Capture webcam frame.
        if (webCamTextureToMatHelper.IsPlaying())
        {

            Mat rgbaMat = webCamTextureToMatHelper.GetMat();

            faceRectInMask = DetectFace(rgbaMat);
            if (faceRectInMask.width == 0 && faceRectInMask.height == 0)
            {
                Debug.Log("A face could not be detected from the input image.");
                return;
            }

            Rect rect = new Rect((int)faceRectInMask.x, (int)faceRectInMask.y,
(int)faceRectInMask.width, (int)faceRectInMask.height);
            rect.inflate(rect.x / 5, rect.y / 5);
            rect = rect.intersect(new Rect(0, 0, rgbaMat.width(), rgbaMat.height()));

            faceMaskTexture = new Texture2D(rect.width, rect.height,
TextureFormat.RGBA32, false);
            faceMaskMat = new Mat(rgbaMat, rect).clone();
            OpenCVForUnity.UnityUtils.Utils.matToTexture2D(faceMaskMat,
faceMaskTexture);
            Debug.Log("faceMaskMat ToString " + faceMaskMat.ToString());

            faceRectInMask = DetectFace(faceMaskMat);
            faceLandmarkPointsInMask =
DetectFaceLandmarkPoints(faceMaskMat, faceRectInMask);

            if (extendForehead)
            {
                AddForeheadPoints(faceLandmarkPointsInMask);
            }

            if (faceRectInMask.width == 0 && faceRectInMask.height == 0)
            {
                RemoveFaceMask();
                Debug.Log("A face could not be detected from the input image.");
```

```
        }
    }
}

/// <summary>
/// Raises the remove face mask button click event.
/// </summary>
public void OnRemoveFaceMaskButtonClick()
{
    RemoveFaceMask();
}

private void RemoveFaceMask()
{
    faceMaskTexture = null;
    if (faceMaskMat != null)
    {
        faceMaskMat.Dispose();
        faceMaskMat = null;
    }

    rectangleTracker.Reset();
    meshOverlay.Reset();
}

private UnityEngine.Rect DetectFace(Mat mat)
{
    if (useDlibFaceDetecter)
    {
        OpenCVForUnityUtils.SetImage(faceLandmarkDetector, mat);
        List<UnityEngine.Rect> result = faceLandmarkDetector.Detect();
        if (result.Count >= 1)
            return result[0];
    }
    else
    {

        using (Mat grayMat = new Mat())
        using (Mat equalizeHistMat = new Mat())
        using (MatOfRect faces = new MatOfRect())
        {
            // convert image to greyscale.
            Imgproc.cvtColor(mat, grayMat, Imgproc.COLOR_RGBA2GRAY);
            Imgproc.equalizeHist(grayMat, equalizeHistMat);
```

```
                cascade.detectMultiScale(equalizeHistMat, faces, 1.1f, 2, 0 |
Objdetect.CASCADE_SCALE_IMAGE, new Size(equalizeHistMat.cols() * 0.15,
equalizeHistMat.cols() * 0.15), new Size());

                List<Rect> faceList = faces.toList();
                if (faceList.Count >= 1)
                {
                    UnityEngine.Rect r = new UnityEngine.Rect(faceList[0].x,
faceList[0].y, faceList[0].width, faceList[0].height);
                    // corrects the deviation of a detection result between OpenCV and
Dlib.
                    r.y += (int)(r.height * 0.1f);
                    return r;
                }
            }
        }
        return new UnityEngine.Rect();
    }

    private List<Vector2> DetectFaceLandmarkPoints(Mat mat,
UnityEngine.Rect rect)
    {
        OpenCVForUnityUtils.SetImage(faceLandmarkDetector, mat);
        List<Vector2> points = faceLandmarkDetector.DetectLandmark(rect);

        return points;
    }

    private void ExtendForehead(Mesh mesh)
    {
        if (mesh.vertices.Length != 68)
            throw new ArgumentException("Invalid face mask mesh", "mesh");

        List<Vector2> verticesList = new List<Vector2>(mesh.vertices.Length);
        foreach (Vector3 v in mesh.vertices)
        {
            verticesList.Add(new Vector2(v.x, v.y));
        }

        AddForeheadPoints(verticesList);

        Vector3[] vertices = new Vector3[verticesList.Count];
        for (int i = 0; i < vertices.Length; i++)
        {
            vertices[i] = new Vector3(verticesList[i].x, verticesList[i].y);
        }
```

```
            mesh.vertices = vertices;

            int[] foreheadTriangles = new int[13 * 3] {
               68, 16, 26,
               68, 26, 25,
               69, 68, 25,
               69, 25, 24,
               69, 24, 23,
               70, 69, 23,

               70, 23, 20,

               71, 70, 20,
               71, 20, 19,
               71, 19, 18,
               72, 71, 18,
               72, 18, 17,
               72, 17, 0
            };
            int[] triangles = new int[mesh.triangles.Length +
foreheadTriangles.Length];
            mesh.triangles.CopyTo(triangles, 0);
            foreheadTriangles.CopyTo(triangles, mesh.triangles.Length);
            mesh.triangles = triangles;

            Vector2[] uv = new Vector2[mesh.uv.Length];
            for (int j = 0; j < uv.Length; j++)
            {
               uv[j].x = vertices[j].x + 0.5f;
               uv[j].y = vertices[j].y + 0.5f;
            }
            mesh.uv = uv;
            mesh.uv2 = (Vector2[])uv.Clone();

            mesh.RecalculateNormals();
         }

         private void AddForeheadPoints(List<Vector2> points)
         {
            if (points.Count != 68)
               throw new ArgumentException("Invalid face landmark points",
"points");

            Vector2 noseLength = new Vector2(points[27].x - points[30].x,
points[27].y - points[30].y);
```

```
        Vector2 glabellaPoint = new Vector2((points[19].x + points[24].x) / 2f,
(points[19].y + points[24].y) / 2f);

        points.Add(new Vector2(points[26].x + noseLength.x * 0.8f, points[26].y
+ noseLength.y * 0.8f));
        points.Add(new Vector2(points[24].x + noseLength.x, points[24].y +
noseLength.y));
        points.Add(new Vector2(glabellaPoint.x + noseLength.x * 1.1f,
glabellaPoint.y + noseLength.y * 1.1f));
        points.Add(new Vector2(points[19].x + noseLength.x, points[19].y +
noseLength.y));
        points.Add(new Vector2(points[17].x + noseLength.x * 0.8f, points[17].y
+ noseLength.y * 0.8f));
    }

    private static void DrawFaceLandmark(Mat imgMat, List<Vector2> points,
Scalar color, int thickness)
    {
        if (points.Count == 73)
        { // If landmark points of forehead exists.
            for (int i = 1; i <= 16; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);

            for (int i = 28; i <= 30; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);

            for (int i = 18; i <= 21; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);
            for (int i = 23; i <= 26; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);
            for (int i = 31; i <= 35; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);
            Imgproc.line(imgMat, new Point(points[30].x, points[30].y), new
Point(points[35].x, points[35].y), color, thickness);

            for (int i = 37; i <= 41; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);
            Imgproc.line(imgMat, new Point(points[36].x, points[36].y), new
Point(points[41].x, points[41].y), color, thickness);
```

```
            for (int i = 43; i <= 47; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);
            Imgproc.line(imgMat, new Point(points[42].x, points[42].y), new
Point(points[47].x, points[47].y), color, thickness);

            for (int i = 49; i <= 59; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);
            Imgproc.line(imgMat, new Point(points[48].x, points[48].y), new
Point(points[59].x, points[59].y), color, thickness);

            for (int i = 61; i <= 67; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), color, thickness);
            Imgproc.line(imgMat, new Point(points[60].x, points[60].y), new
Point(points[67].x, points[67].y), color, thickness);

            for (int i = 69; i <= 72; ++i)
                Imgproc.line(imgMat, new Point(points[i].x, points[i].y), new
Point(points[i - 1].x, points[i - 1].y), new Scalar(0, 255, 0, 255), thickness);
        }
        else
        {
            OpenCVForUnityUtils.DrawFaceLandmark(imgMat, points, color,
thickness);
        }
    }

    private Texture2D CreateFaceMaskAlphaMaskTexture(float width, float
height, Vector2[] uv, bool makeBothEyesTransparent = true, bool
makeMouthTransparent = true)
    {
        if (uv.Length != 68 && uv.Length != 73)
            throw new ArgumentException("Invalid face landmark points", "uv");

        Vector2[] facialContourUVPoints = new Vector2[0];
        if (uv.Length == 68)
        {
            facialContourUVPoints = new Vector2[] {
                uv [0],
                uv [1],
                uv [2],
                uv [3],
                uv [4],
                uv [5],
```

```
            uv [6],
            uv [7],
            uv [8],
            uv [9],
            uv [10],
            uv [11],
            uv [12],
            uv [13],
            uv [14],
            uv [15],
            uv [16],
            uv [26],
            uv [25],
            uv [24],
            uv [23],
            uv [20],
            uv [19],
            uv [18],
            uv [17]
        };
    }
    else if (uv.Length == 73)
    { // If landmark points of forehead exists.
        facialContourUVPoints = new Vector2[] {
            uv [0],
            uv [1],
            uv [2],
            uv [3],
            uv [4],
            uv [5],
            uv [6],
            uv [7],
            uv [8],
            uv [9],
            uv [10],
            uv [11],
            uv [12],
            uv [13],
            uv [14],
            uv [15],
            uv [16],
            uv [68],
            uv [69],
            uv [70],
            uv [71],
            uv [72]
```

```
            };
        }

        Vector2[] rightEyeContourUVPoints = new Vector2[] {
            uv [36],
            uv [37],
            uv [38],
            uv [39],
            uv [40],
            uv [41]
        };

        Vector2[] leftEyeContourUVPoints = new Vector2[] {
            uv [42],
            uv [43],
            uv [44],
            uv [45],
            uv [46],
            uv [47]
        };

        Vector2[] mouthContourUVPoints = new Vector2[] {
            uv [60],
            uv [61],
            uv [62],
            uv [63],
            uv [64],
            uv [65],
            uv [66],
            uv [67]
        };

        Vector2[][] exclusionAreas;
        if (makeBothEyesTransparent == true && makeMouthTransparent ==
false)
        {
            exclusionAreas = new Vector2[][] { rightEyeContourUVPoints,
leftEyeContourUVPoints };
        }
        else if (makeBothEyesTransparent == false && makeMouthTransparent
== true)
        {
            exclusionAreas = new Vector2[][] { mouthContourUVPoints };
        }
        else if (makeBothEyesTransparent == true && makeMouthTransparent ==
true)
```

```
            {
                exclusionAreas = new Vector2[][] {
                    rightEyeContourUVPoints,
                    leftEyeContourUVPoints,
                    mouthContourUVPoints
                };
            }
            else
            {
                exclusionAreas = new Vector2[][] { };
            }

            return AlphaMaskTextureCreater.CreateAlphaMaskTexture(width, height,
facialContourUVPoints, exclusionAreas);
        }
    }
}
```

3. FaceMaskColor

```
using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;
using OpenCVForUnity.CoreModule;
using OpenCVForUnity.ImgprocModule;
using Rect = OpenCVForUnity.CoreModule.Rect;

namespace FaceMaskExample
{
    public class FaceMaskColorCorrector
    {
        Mat src_mask;
        Mat dst_mask;
        Dictionary<int, Texture2D> LUTTexDict;

        Point[] src_facialContourPoints;
        Point[] dst_facialContourPoints;

        public FaceMaskColorCorrector()
        {
            LUTTexDict = new Dictionary<int, Texture2D>();

            src_facialContourPoints = new Point[9];
            for (int i = 0; i < src_facialContourPoints.Length; i++)
            {
                src_facialContourPoints[i] = new Point();
```

```
        }

        dst_facialContourPoints = new Point[9];
        for (int i = 0; i < dst_facialContourPoints.Length; i++)
        {
            dst_facialContourPoints[i] = new Point();
        }
    }

    public virtual void CreateLUTTex(int id)
    {
        if (!LUTTexDict.ContainsKey(id))
            LUTTexDict.Add(id, new Texture2D(256, 1, TextureFormat.RGB24,
false));
    }

    public virtual Texture2D UpdateLUTTex(int id, Mat src, Mat dst,
List<Vector2> src_landmarkPoints, List<Vector2> dst_landmarkPoints)
    {
        if (src_mask != null && (src.width() != src_mask.width() || src.height() !=
src_mask.height()))
        {
            src_mask.Dispose();
            src_mask = null;
        }
        src_mask = src_mask ?? new Mat(src.rows(), src.cols(),
CvType.CV_8UC1, Scalar.all(0));

        if (dst_mask != null && (dst.width() != dst_mask.width() || dst.height() !=
dst_mask.height()))
        {
            dst_mask.Dispose();
            dst_mask = null;
        }
        dst_mask = dst_mask ?? new Mat(dst.rows(), dst.cols(),
CvType.CV_8UC1, Scalar.all(0));

        // Get facial contour points.
        GetFacialContourPoints(src_landmarkPoints, src_facialContourPoints);
        GetFacialContourPoints(dst_landmarkPoints, dst_facialContourPoints);

        // Get facial contour rect.
        Rect src_facialContourRect = Imgproc.boundingRect(new
MatOfPoint(src_facialContourPoints));
        Rect dst_facialContourRect = Imgproc.boundingRect(new
MatOfPoint(dst_facialContourPoints));
```

```
        src_facialContourRect = src_facialContourRect.intersect(new Rect(0, 0,
src.width(), src.height()));
        dst_facialContourRect = dst_facialContourRect.intersect(new Rect(0, 0,
dst.width(), dst.height()));

        Mat src_ROI = new Mat(src, src_facialContourRect);
        Mat dst_ROI = new Mat(dst, dst_facialContourRect);
        Mat src_mask_ROI = new Mat(src_mask, src_facialContourRect);
        Mat dst_mask_ROI = new Mat(dst_mask, dst_facialContourRect);

        GetPointsInFrame(src_mask_ROI, src_facialContourPoints,
src_facialContourPoints);
        GetPointsInFrame(dst_mask_ROI, dst_facialContourPoints,
dst_facialContourPoints);

        src_mask_ROI.setTo(new Scalar(0));
        dst_mask_ROI.setTo(new Scalar(0));
        Imgproc.fillConvexPoly(src_mask_ROI, new
MatOfPoint(src_facialContourPoints), new Scalar(255));
        Imgproc.fillConvexPoly(dst_mask_ROI, new
MatOfPoint(dst_facialContourPoints), new Scalar(255));

        Texture2D LUTTex;
        if (LUTTexDict.ContainsKey(id))
        {
            LUTTex = LUTTexDict[id];
        }
        else
        {
            LUTTex = new Texture2D(256, 1, TextureFormat.RGB24, false);
            LUTTexDict.Add(id, LUTTex);
        }

        FaceMaskShaderUtils.CalculateLUT(src_ROI, dst_ROI, src_mask_ROI,
dst_mask_ROI, LUTTex);

        return LUTTex;
    }

    public virtual void DeleteLUTTex(int id)
    {
        if (LUTTexDict.ContainsKey(id))
        {
            Texture2D.Destroy(LUTTexDict[id]);
            LUTTexDict.Remove(id);
        }
```

```
    }

    public virtual Texture2D GetLUTTex(int id)
    {
        if (LUTTexDict.ContainsKey(id))
        {
            return LUTTexDict[id];
        }
        return null;
    }

    protected virtual void GetFacialContourPoints(List<Vector2>
landmark_points, Point[] dst_points)
    {
        if (landmark_points.Count < 9)
            throw new ArgumentException("Invalid landmark_points.");

        if (dst_points.Length != 9)
            throw new ArgumentException("Invalid points.");

        dst_points[0].x = landmark_points[0].x;
        dst_points[0].y = landmark_points[0].y;
        dst_points[1].x = landmark_points[3].x;
        dst_points[1].y = landmark_points[3].y;
        dst_points[2].x = landmark_points[5].x;
        dst_points[2].y = landmark_points[5].y;
        dst_points[3].x = landmark_points[8].x;
        dst_points[3].y = landmark_points[8].y;
        dst_points[4].x = landmark_points[11].x;
        dst_points[4].y = landmark_points[11].y;
        dst_points[5].x = landmark_points[13].x;
        dst_points[5].y = landmark_points[13].y;
        dst_points[6].x = landmark_points[16].x;
        dst_points[6].y = landmark_points[16].y;
        float nose_length_x = landmark_points[27].x - landmark_points[30].x;
        float nose_length_y = landmark_points[27].y - landmark_points[30].y;
        dst_points[7].x = landmark_points[26].x + nose_length_x;
        dst_points[7].y = landmark_points[26].y + nose_length_y;
        dst_points[8].x = landmark_points[17].x + nose_length_x;
        dst_points[8].y = landmark_points[17].y + nose_length_y;
    }

    protected virtual void GetPointsInFrame(Mat frame, Point[] points, Point[]
dst_points)
    {
        if (points.Length != dst_points.Length)
```

```
            throw new ArgumentException("points.Length != dst_points.Length");

        Size wholesize = new Size();
        Point ofs = new Point();
        frame.locateROI(wholesize, ofs);

        for (int i = 0; i < points.Length; i++)
        {
            dst_points[i].x = points[i].x - ofs.x;
            dst_points[i].y = points[i].y - ofs.y;
        }
    }

    public virtual void Reset()
    {
        foreach (var key in LUTTexDict.Keys)
        {
            Texture2D.Destroy(LUTTexDict[key]);
        }
        LUTTexDict.Clear();
    }

    public virtual void Dispose()
    {
        if (src_mask != null)
        {
            src_mask.Dispose();
            src_mask = null;
        }

        if (dst_mask != null)
        {
            dst_mask.Dispose();
            dst_mask = null;
        }

        Reset();
    }
  }
}
```

4. FaceMaskShaderUtils

```
using UnityEngine;
using System;
using System.Collections.Generic;
```

```
using System.Collections;
using OpenCVForUnity.CoreModule;
using OpenCVForUnity.UnityUtils;
using OpenCVForUnity.UtilsModule;

namespace FaceMaskExample
{
   public class FaceMaskShaderUtils
   {
      // Match histograms of 'src' to that of 'dst', according to both masks.
      public static void CalculateLUT(Mat src, Mat dst, Mat src_mask, Mat
dst_mask, Texture2D LUTTex)
      {
         if (src.channels() < 3)
            throw new ArgumentException("src.channels() < 3");

         if (dst.channels() < 3)
            throw new ArgumentException("dst.channels() < 3");

         if (src_mask.channels() != 1)
            throw new ArgumentException("src_mask.channels() != 1");

         if (dst_mask.channels() != 1)
            throw new ArgumentException("dst_mask.channels() != 1");

         if (src_mask != null && src.total() != src_mask.total())
            throw new ArgumentException("src.total() != src_mask.total()");

         if (dst_mask != null && dst.total() != dst_mask.total())
            throw new ArgumentException("dst.total() != dst_mask.total()");

         if (LUTTex.width != 256 || LUTTex.height != 1 || LUTTex.format !=
TextureFormat.RGB24)
            throw new ArgumentException("Invalid LUTTex.");

         byte[] LUT = new byte[100 * 256];
         double[][] src_hist = new double[100][];
         for (int i = 0; i < src_hist.Length; i++)
         {
            src_hist[i] = new double[256];
         }
         double[][] dst_hist = new double[100][];
         for (int i = 0; i < dst_hist.Length; i++)
         {
            dst_hist[i] = new double[256];
         }
```

```
double[][] src_cdf = new double[100][];
for (int i = 0; i < src_cdf.Length; i++)
{
   src_cdf[i] = new double[256];
}
double[][] dst_cdf = new double[100][];
for (int i = 0; i < dst_cdf.Length; i++)
{
   dst_cdf[i] = new double[256];
}

double[] src_histMax = new double[100];
double[] dst_histMax = new double[100];

byte[] src_mask_byte = null;
byte[] dst_mask_byte = null;
if (src_mask != null)
{
   src_mask_byte = new byte[src_mask.total() * src_mask.channels()];
   MatUtils.copyFromMat<byte>(src_mask, src_mask_byte);
}
if (dst_mask != null)
{
   dst_mask_byte = new byte[dst_mask.total() * dst_mask.channels()];
   MatUtils.copyFromMat<byte>(dst_mask, dst_mask_byte);
}

byte[] src_byte = new byte[src.total() * src.channels()];
MatUtils.copyFromMat<byte>(src, src_byte);
byte[] dst_byte = new byte[dst.total() * dst.channels()];
MatUtils.copyFromMat<byte>(dst, dst_byte);

int pixel_i = 0;
int channels = src.channels();
int total = (int)src.total();
if (src_mask_byte != null)
{
   for (int i = 0; i < total; i++)
   {
      if (src_mask_byte[i] != 0)
      {
         byte c = src_byte[pixel_i];
         src_hist[0][c]++;
         if (src_hist[0][c] > src_histMax[0])
            src_histMax[0] = src_hist[0][c];
```

```
            c = src_byte[pixel_i + 1];
            src_hist[1][c]++;
            if (src_hist[1][c] > src_histMax[1])
               src_histMax[1] = src_hist[1][c];

            c = src_byte[pixel_i + 2];
            src_hist[2][c]++;
            if (src_hist[2][c] > src_histMax[2])
               src_histMax[2] = src_hist[2][c];
         }

         // Advance to next pixel
         pixel_i += channels;
      }
   }
   else
   {
      for (int i = 0; i < total; i++)
      {
         byte c = src_byte[pixel_i];
         src_hist[0][c]++;
         if (src_hist[0][c] > src_histMax[0])
            src_histMax[0] = src_hist[0][c];

         c = src_byte[pixel_i + 1];
         src_hist[1][c]++;
         if (src_hist[1][c] > src_histMax[1])
            src_histMax[1] = src_hist[1][c];

         c = src_byte[pixel_i + 2];
         src_hist[2][c]++;
         if (src_hist[2][c] > src_histMax[2])
            src_histMax[2] = src_hist[2][c];

         // Advance to next pixel
         pixel_i += channels;
      }
   }

   pixel_i = 0;
   channels = dst.channels();
   total = (int)dst.total();
   if (dst_mask_byte != null)
   {
      for (int i = 0; i < total; i++)
      {
```

```
        if (dst_mask_byte[i] != 0)
        {
            byte c = dst_byte[pixel_i];
            dst_hist[0][c]++;
            if (dst_hist[0][c] > dst_histMax[0])
                dst_histMax[0] = dst_hist[0][c];

            c = dst_byte[pixel_i + 1];
            dst_hist[1][c]++;
            if (dst_hist[1][c] > dst_histMax[1])
                dst_histMax[1] = dst_hist[1][c];

            c = dst_byte[pixel_i + 2];
            dst_hist[2][c]++;
            if (dst_hist[2][c] > dst_histMax[2])
                dst_histMax[2] = dst_hist[2][c];
        }
        // Advance to next pixel
        pixel_i += channels;
    }
}
else
{
    for (int i = 0; i < total; i++)
    {
        byte c = dst_byte[pixel_i];
        dst_hist[0][c]++;
        if (dst_hist[0][c] > dst_histMax[0])
            dst_histMax[0] = dst_hist[0][c];

        c = dst_byte[pixel_i + 1];
        dst_hist[1][c]++;
        if (dst_hist[1][c] > dst_histMax[1])
            dst_histMax[1] = dst_hist[1][c];

        c = dst_byte[pixel_i + 2];
        dst_hist[2][c]++;
        if (dst_hist[2][c] > dst_histMax[2])
            dst_histMax[2] = dst_hist[2][c];

        // Advance to next pixel
        pixel_i += channels;
    }
}

//normalize hist
```

```
for (int i = 0; i < 256; i++)
{
   src_hist[0][i] /= src_histMax[0];
   src_hist[1][i] /= src_histMax[1];
   src_hist[2][i] /= src_histMax[2];

   dst_hist[0][i] /= dst_histMax[0];
   dst_hist[1][i] /= dst_histMax[1];
   dst_hist[2][i] /= dst_histMax[2];
}

// Calc cumulative distribution function (CDF)
src_cdf[0][0] = src_hist[0][0];
src_cdf[1][0] = src_hist[1][0];
src_cdf[2][0] = src_hist[2][0];
dst_cdf[0][0] = dst_hist[0][0];
dst_cdf[1][0] = dst_hist[1][0];
dst_cdf[2][0] = dst_hist[2][0];
for (int i = 1; i < 256; i++)
{
   src_cdf[0][i] = src_cdf[0][i - 1] + src_hist[0][i];
   src_cdf[1][i] = src_cdf[1][i - 1] + src_hist[1][i];
   src_cdf[2][i] = src_cdf[2][i - 1] + src_hist[2][i];

   dst_cdf[0][i] = dst_cdf[0][i - 1] + dst_hist[0][i];
   dst_cdf[1][i] = dst_cdf[1][i - 1] + dst_hist[1][i];
   dst_cdf[2][i] = dst_cdf[2][i - 1] + dst_hist[2][i];
}

// Normalize CDF
for (int i = 0; i < 256; i++)
{
   src_cdf[0][i] /= src_cdf[0][255];
   src_cdf[1][i] /= src_cdf[1][255];
   src_cdf[2][i] /= src_cdf[2][255];

   dst_cdf[0][i] /= dst_cdf[0][255];
   dst_cdf[1][i] /= dst_cdf[1][255];
   dst_cdf[2][i] /= dst_cdf[2][255];
}

// Create lookup table
const double HISTMATCH_EPSILON = 0.000001f;
for (int i = 0; i < 3; i++)
{
   int last = 0;
```

```
            for (int j = 0; j < 256; j++)
            {
               double F1j = src_cdf[i][j];

               for (int k = last; k < 256; k++)
               {
                  double F2k = dst_cdf[i][k];
                  if (Math.Abs(F2k - F1j) < HISTMATCH_EPSILON || F2k > F1j)
                  {
                     LUT[(j * 3) + i] = (byte)k;
                     last = k;
                     break;
                  }
               }
            }
         }

         LUTTex.LoadRawTextureData(LUT);
         LUTTex.Apply(false);
      }
   }
}
```

5. FaceMaskData

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class FaceMaskData : MonoBehaviour
{
   [SerializeField]
   private Texture2D _image;

   public Texture2D image
   {
      get { return this._image; }
      set { this._image = value; }
   }

   /// <summary>
   /// Determines if to use dynamically detected points.
   /// </summary>
   [TooltipAttribute("Determines if to use dynamically detected points.")]
   public bool isDynamicMode = true;
```

```
/// <summary>
/// Determines if to enable color correction.
/// </summary>
[TooltipAttribute("Determines if to enable color correction.")]
public bool enableColorCorrection = true;

[SerializeField]
private Rect _faceRect = new Rect(78, 95, 151, 150);

public Rect faceRect
{
   get { return this._faceRect; }
   set { this._faceRect = value; }
}

[SerializeField]
private List<Vector2> _landmarkPoints = new List<Vector2>() {
   new Vector2 (84, 148),
   new Vector2 (84, 167),
   new Vector2 (86, 187),
   new Vector2 (89, 206),
   new Vector2 (96, 224),
   new Vector2 (106, 240),
   new Vector2 (119, 253),
   new Vector2 (134, 264),
   new Vector2 (151, 266),
   new Vector2 (168, 264),
   new Vector2 (184, 254),
   new Vector2 (197, 241),
   new Vector2 (207, 226),
   new Vector2 (214, 209),
   new Vector2 (218, 190),
   new Vector2 (221, 170),
   new Vector2 (221, 150),
   new Vector2 (100, 125),
   new Vector2 (108, 117),
   new Vector2 (119, 114),
   new Vector2 (130, 116),
   new Vector2 (141, 120),
   new Vector2 (164, 120),
   new Vector2 (176, 116),
   new Vector2 (187, 115),
   new Vector2 (199, 118),
   new Vector2 (206, 126),
   new Vector2 (153, 133),
   new Vector2 (153, 144),
```

```
            new Vector2 (154, 156),
            new Vector2 (154, 168),
            new Vector2 (142, 181),
            new Vector2 (148, 182),
            new Vector2 (154, 184),
            new Vector2 (160, 182),
            new Vector2 (165, 181),
            new Vector2 (113, 134),
            new Vector2 (120, 127),
            new Vector2 (129, 127),
            new Vector2 (136, 136),
            new Vector2 (128, 137),
            new Vector2 (119, 137),
            new Vector2 (170, 137),
            new Vector2 (177, 128),
            new Vector2 (187, 128),
            new Vector2 (193, 136),
            new Vector2 (188, 139),
            new Vector2 (178, 138),
            new Vector2 (127, 215),
            new Vector2 (135, 204),
            new Vector2 (145, 199),
            new Vector2 (154, 201),
            new Vector2 (163, 199),
            new Vector2 (173, 205),
            new Vector2 (178, 218),
            new Vector2 (173, 225),
            new Vector2 (163, 229),
            new Vector2 (154, 230),
            new Vector2 (144, 228),
            new Vector2 (134, 224),
            new Vector2 (131, 215),
            new Vector2 (145, 206),
            new Vector2 (154, 207),
            new Vector2 (163, 207),
            new Vector2 (175, 217),
            new Vector2 (163, 219),
            new Vector2 (154, 220),
            new Vector2 (144, 218)
        };
        public List<Vector2> landmarkPoints
        {
            get { return this._landmarkPoints; }
            set { this._landmarkPoints = value; }
        }
    }
```